BORG: Block-reORGanization for Self-optimizing Storage Systems

Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett†, Jason Liptak‡,
Raju Rangaswami, and Vagelis Hristidis
Florida International University, †Carnegie Mellon University, ‡Syracuse University

Abstract

This paper presents the design, implementation, and evaluation of BORG, a self-optimizing storage system that performs *automatic block reorganization* based on the observed I/O workload. BORG is motivated by three characteristics of I/O workloads: non-uniform access frequency distribution, temporal locality, and partial determinism in non-sequential accesses. To achieve its objective, BORG manages a small, dedicated partition on the disk drive, with the goal of servicing majority of the I/O requests from within this partition with significantly reduced seek and rotational delays. BORG remains oblivious to the rest of the storage stack, including applications, file system(s), and I/O schedulers, thereby requiring no or minimal modification to storage stack implementations. We evaluated a Linux implementation of BORG using several real-world workloads, including individual user desktop environments, a web-server, a virtual machine monitor, and an SVN server. These experiments demonstrate BORG's effectiveness in improving I/O performance and its incurred resource overhead.

1 Introduction

There is a continual increase in the gap between CPU performance and disk drive performance. While the steady increase in main memory sizes attempts to bridge this gap, the impact is relatively small; Patterson *et al.* [23] have pointed out that disk drive capacities and workload working-set sizes tend to grow at a faster rate than memory sizes. Present day file systems, which control space allocation on the disk drive, employ static data layouts [7, 14, 18, 20, 35, 4]. Mostly, they aim to preserve the directory structure of the file system and optimize for sequential access to entire files. No file system today takes into account the dynamic characteristics of I/O workload within its data management mechanisms.

We conducted experiments to reconcile past observations about the nature of I/O workloads [28, 6, 8] in the context of current-day systems including end-user and server-class systems. Our key observations that motivate BORG are: *(i)* on-disk data exhibit a *non-uniform access frequency distribution*; the "frequently accessed" data is usually a small fraction of the total data stored when considering a coarse-granularity time-frame. *(ii)* considering a fine-granularity time-frame, the "on-disk working-set" of typical I/O workloads is dynamic; nevertheless, workloads exhibit *temporal locality* in the data that they access. *(iii)* I/O workloads exhibit *partial determinism* in their disk access patterns; besides sequential accesses to portions of files, fragments of the block access sequence that lead to non-sequential[1] disk accesses also repeat. We elaborate on these observations in § 2.

While the above observations mostly validate the prior studies, and may even appear largely intuitive, surprisingly, there is a lack of commodity storage systems that utilize these observations to reduce I/O times. We believe that such systems do not exist because *(i)* key design and implementation issues related to the feasibility of such systems have not been resolved, and *(ii)* the scope of effectiveness of such systems has not been determined.

We built BORG, an online *Block-reORGanizing* storage system to comprehensively address the above issues. BORG correlates disk blocks based on block access patterns to capture the I/O workload characteristics. BORG manages a dedicated, *OPtimized Target (OPT)* partition. It dynamically copies working-set data blocks (possibly spread over the entire disk) in their relative access sequence contiguously within this partition, thus simultaneously reducing seek and rotational delays. In addition, it assimilates all *write requests* into the OPT partition's write buffer. Since BORG operates in the background it presents little interference to foreground applications. Also, BORG provides strong block-layer data consistency to upper layers, by maintaining a persistent page-level *indirection map*.

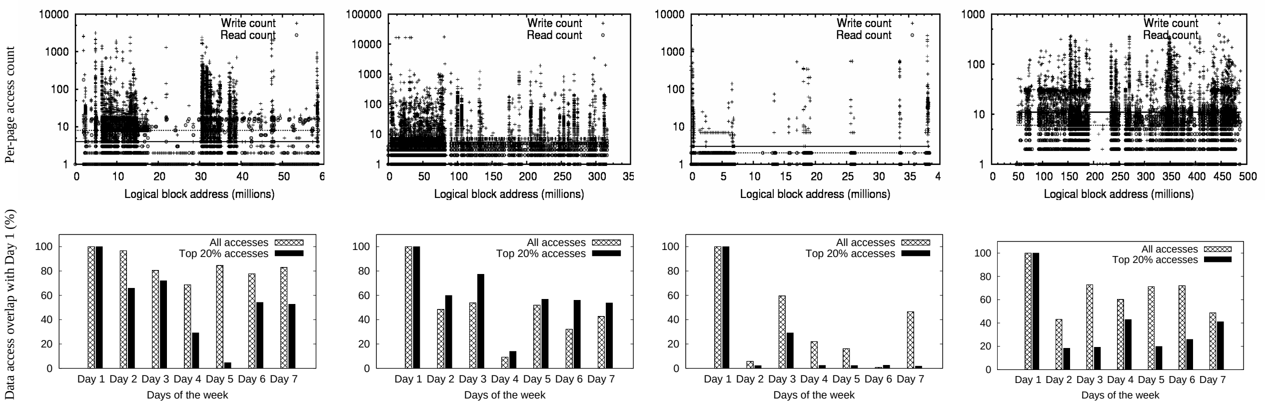
We evaluated a Linux implementation of BORG for a variety of workloads including a development workstation, an SVN server, a web server, a virtual machine monitor, as well as several individual desktop applications. The evaluation shows both the benefits and shortcomings of BORG as well as its resource overheads. Particularly, BORG can degrade performance when a non-sequential read workload suddenly shifts its on-disk working-set. For most workloads, however, BORG increased average case disk throughput in the range 13.3% to 50%, offering the greatest benefit in the case of non-sequential write workloads. A sensitivity study with var-

---

[1] We use the term "non-sequential I/O" in a slightly different sense than "random I/O" since by definition true random I/O may not exhibit repeatable determinism.

Workload type	File System size [GB]	Memory size [GB]	Reads [GB]		Writes [GB]		File System accessed	Top 20% data access	Partial determinism
			Total	Unique	Total	Unique			
office	8.29	1.5	6.49	1.63	0.32	0.22	22.22 %	51.40 %	65.42 %
developer	45.59	2.0	3.82	2.57	10.46	3.96	14.32 %	60.27 %	61.56 %
SVN server	23.96	0.5	0.29	0.17	0.62	0.18	1.41 %	45.79 %	50.73 %
web server	169.54	0.5	21.07	7.32	2.24	0.33	4.51 %	59.50 %	15.55 %

Table 1: **Summary statistics of week-long traces obtained from four different systems.**



(a) office (b) developer (c) SVN server (d) web server

Figure 1: **Frequency and working-set plots for week-long traces from four different systems.** *In the top row graphs, the solid and dashed lines represent the top 20% threshold access counts for writes and reads respectively.*

ious parameters of BORG demonstrates the importance of careful parameter choice; a self-configuring BORG is certainly a logical and feasible direction. Memory overheads of BORG are bound within 0.25% of OPT, but CPU overheads are higher. Fortunately, most processing can be done in the background and there is ample room for improvement.

This paper makes the following contributions: *(i)* we study the characteristics of I/O workloads and show how the findings motivate BORG (§ 2) , *(ii)* we motivate and present the detailed design and the first implementation of a disk data re-organizing system that adapts itself to changes in the I/O workload (§ 3 and § 4), *(iii)* we present the challenges faced in building such a system and our solutions to it (§ 5), and finally, *(iv)* we evaluate the system to quantify its merits and weaknesses (§ 6).

2 Characteristics of I/O Workloads

In this section, we investigate the characteristics of modern I/O workloads, specifically elaborating on those that directly motivate BORG. We collected I/O traces, downstream of an active page cache, over a one-week period from four different machines. These machines have different I/O workloads, including an*office-class* and *developer-class* desktop workloads, a small-scale version control *SVN (Subversion) server* of our research group, and our department's *production web-server*. The traces are summarized in Table 1. We define the *on-disk working-set*[2] of an I/O workload as the set of all unique blocks accessed in a given interval.

2.1 Non-uniform Access Frequency Distribution

Researchers have pointed out that file system data have non-uniform access frequency distribution [1, 36, 27]. This was confirmed in the traces that we collected where less than 1.5-22.3% of the file systems were accessed over the duration of an entire week. Figure 1 (top row) shows block access frequency plots for the workloads. Some uniform trends to be observed are that while the really high frequency accesses tend to be writes, there are a substantial number of reads that occur repeatedly (some as many as 100 times). We also observed a skewness in data access behavior. As depicted in Table 1, the top 20% most frequently accessed blocks contributed to a substantially large (∼45-66%) percentage of the total accesses. These numbers are within the ranges reported by Gómez and Santonja (Figure 2(a) in [6]) for the Cello traces they examined.

Based on the above observations, it is reasonable to expect that co-locating frequently accessed data in a small area of the disk would help reduce seek times, as compared to when the same data is spread throughout the entire disk area. Akyurek et. al. [1] have demonstrated the performance benefits of such an optimization via a simulation study. This observation also motivates the choice of reorganizing copies of "popular blocks" in BORG.

---

[2] henceforth also referred to simply as "working-set".

2.2 Temporal Locality

Temporal locality in I/O workloads is observed when the on-disk working-sets remain mostly static over short durations. Here, we refer to a locality of hours, days, or weeks, rather than seconds or minutes (typical of main memory accesses). For instance, a developer may work on a few projects over a period of a few weeks or months, typically resulting in her daily or weekly working sets being substantially smaller than her entire disk size. In servers, popularity of client requests result in temporal locality. A web server's top-level links tend to be accessed more frequently than content that is embedded much deeper in the web-site; an important new revision of a specific repository on an SVN server is likely to be accessed repeatedly over the initial weeks.

Figure 1 (bottom row) depicts the changes in the per-day working-sets of the I/O workload. The two end-user I/O workloads and the web server workload exhibit large overlaps in the data accessed across successive days of the week-long trace with the first day of the trace. There is substantial overlap even among the top 20% most accessed data across successive days. Interestingly, these workloads do not necessarily exhibit a gradual decay in working-set overlap with day 1 as one might expect, indicating that popularity is consistent across multi-day periods. The SVN server exhibits anomalous behavior because periods of high *commit* activity degrade temporal locality (new data gets created), while periods of high *update* activity improve temporal locality.

These observations indicate that optimizing layout based on past I/O activity can improve future I/O performance for some workloads. This motivates planning block reorganization based on past activity in BORG.

2.3 Partial Determinism

Partial determinism in I/O workload occurs when certain non-sequential accesses in the block access sequence are found to repeat. A *non-sequential access* is defined by a sequence of two I/O operations that are addressed non-contiguous block addresses. It manifests in both end-user systems and servers. For instance, I/O during application start-up is largely deterministic, both in terms of the set of I/O requests and the sequence in which they are requested. Reading files related to a repeatable task such as setting up a project in an integrated development environment, compilation, linking, word-processing, etc. result in a deterministic I/O pattern. In a web-server, accessing a web-page involves accessing associated sub-pages, images, scripts, etc., in deterministic order.

In Table 1, we present the *partial determinism* for each workload calculated as the percentage of non-sequential accesses that repeat at least once during the week. The partial determinism percentages are high for the two end-user and the SVN server workloads. Further, for each of these workloads, there were a non-trivial amount of non-sequential accesses that repeated as many as 100 times. These findings suggest that there is ample scope for optimizing the repeated non-sequential access patterns.

3 Overview and Architecture

BORG is motivated by the simple question: *What storage system optimizations based on workload characteristics can allow applications to utilize the disk drive more efficiently than current systems do?* This section presents the rationale behind the design decisions in BORG and its system architecture.

3.1 BORG Design Decisions

A Disk-based Cache.

The operating system uses main memory to cache frequently and recently accessed file system data to reduce the number of disk accesses incurred. In any given duration of time, the effectiveness of the cache is largely dependent on the on-disk working-set of the I/O workload, and can degrade when this working-set increases beyond the size of the page cache. Storage optimizations such as prefetching [31, 15, 22] and I/O scheduling [30, 25, 24, 12] help improve disk I/O performance in such situations.

Using a disk-based cache as an extension of the main memory cache offers three complementary advantages in comparison to main memory caching alone, prefetching, and I/O scheduling. First, it is more effective as a cache (than main memory) because it offers a less expensive (and thus larger) as well as reliable caching solution, thus allowing data to be cache-resident for long periods of time. Second, the size of the disk-based cache can easily be configured by the system administrator without changing any hardware. And finally, dynamically optimizing data layout based on access patterns within a disk-based cache provides the unique ability to make originally non-sequential data accesses more sequential.

A Block Layer Solution.

A self-optimizing storage solution can be built at any layer in the storage stack (shown in Figure 2). Block level attributes of disk I/O operations are not easily obtained at the VFS or the page cache layer. While file system layer solutions can benefit from semantic knowledge of blocks, they incur a significant disadvantage in being tied to a specific file system (and perhaps even version). Device driver encapsulations (interface at P4) are incapable of capturing upper layer attributes, such as process ID and request time-stamp due to I/O scheduler reordering and loss of process context.

We contend that the block layer (interface at P3) is ideal for introducing block reorganization for serveral reasons. First, key temporal, block- and process- level attributes about disk accesses are available. Second, operating at the block layer makes the solution independent of the file system layer above, allowing it the flexibility to support multiple heterogeneous file systems simultaneously. Finally, new abstractions due to virtualization trends (e.g., virtual block device abstraction) as well as network-attached storage environments (SAN and NAS) can be supported in a straightforward way. In the case of SAN, BORG can reside on the client where all context for I/O operations are readily available with the underlying assumption that the SAN device's logical block address space is optimized for sequential access. In the case of NAS, the BORG layer can reside within the NAS device where I/O context is readily available. Modifying the NAS interface to include process associations within file I/O requests can complete the profile information.

Using an Independent OPT partition.
The file system optimizes for sequential accesses to entire files, a common form of file access. However, certain workloads, including application start-up, content indexing and, web-page requests, exhibit a more non-sequential, but deterministic, access behavior. It is thus possible that the same set of data can be accessed sequentially by some applications and non-sequentially by others. Further, some deterministic non-sequential accesses may only be temporary phenomenon.

Based on this observation, Akyurek and Salem [1] have argued in favor of *copying* rather than *shuffling* [36, 27] of data. Copying retains original sequential layouts so a choice of location based on the observed access pattern may be possible. Reverting back to the original layout is straightforward. Borrowing the same principle, rather than permanently disturbing the sequential layout of files, BORG operates on copies of blocks placed temporarily in an independent OPT partition, optimizing for the current common case of access for each data block.

3.2 BORG Architecture

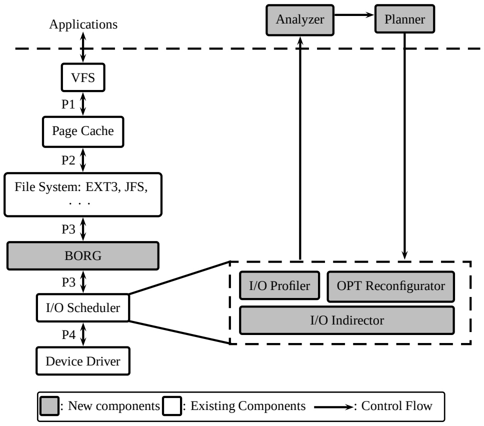


Figure 2: **BORG System Architecture.**

Abstractly, BORG follows a four-stage process:

1. *profiling* application block I/O accesses,
2. *analyzing* I/O accesses to derive access patterns,
3. *planning* a modification to the data layout, and
4. *executing* the plan to reconfigure the data layout.

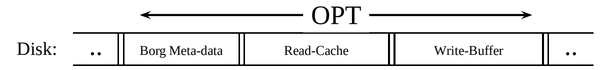
In addition, an I/O indirection mechanism runs continuously re-directing requests to the partition that it optimizes as required. Figure 2 presents the architecture of BORG in relation to the storage stack within the operating system. The modification to the existing storage stack is in the form of a new layer, which we term *BORG layer*, that implements three major components: the *I/O profiler*, the *OPT reconfigurator* and the *I/O Indirector*. A secondary throttle-friendly user-space component implements the *analyzer* and the *planner* stages of BORG and performs computation and memory-intensive tasks. While profiling and indirection are both continuous processes, the other stages run periodically and in succession culminating in a reconfiguration operation.

For the I/O profiler, we use a low-overhead kernel tool called `blktrace` [2]. The analyzer reads the I/O trace collected by the profiler periodically (based on a configurable reconfiguration interval) and derives data access patterns. Subsequently, the planner uses these data access patterns and generates a new reconfiguration plan for the OPT partition, which it communicates to the OPT reconfigurator component. The user-space analyzer and planner components run as a low-priority process, utilizing only otherwise free system resources. Under heavy system load, the only impact to BORG is that generating the new reconfiguration plan would be delayed.

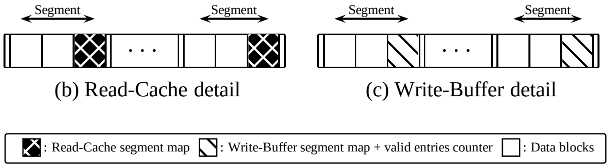
The OPT reconfigurator is responsible for the periodic reconfiguration of the OPT partition, per the *layout plan* specified by the planner. The reconfigurator issues low-priority disk I/Os to accomplish its task, minimizing the interference to foreground disk accesses. Finally, the I/O indirector continuously directs I/O requests either to the FS partition or the OPT partition, based on the specifics of the request and the contents of the OPT.

3.3 OPT Space Management

The OPtimized Target partition (OPT) as managed by BORG is shown in Figure 3. To reduce head movement, we suggest that the OPT partition be created adjoining the *swap* partition if virtual memory is used. BORG partitions the OPT into three fragments: *BORG Meta-data*, *Read-cache*, and *Write-buffer*. The Read-cache and Write-buffer are further sub-divided into fixed-length segments which store both data and (valid/invalid) map entries for the segment. The in-memory indirection map (elaborated in § 4.5) maintained by BORG is a union of all the segment map entries in the OPT. The OPT map entries are synchronously updated each time the in-memory map information changes. Additionally,



(a) OPT overview



(b) Read-Cache detail (c) Write-Buffer detail

Figure 3: **Format of the OPT partition.** *Each entry in the Write-Buffer and Read-Cache map tables is a 3-tuple of the form (FS LBA, OPT LBA, valid bit).*

the segment map in the write-buffer contains a "valid entries counter" to track space usage in the write buffer.

Magic number	BORG OPTpartition identifier.
`BORG_REQUIRE` bit	OPT contains dirty data.
OPT size	OPT partition size.
Read-cache info	Offset and size of the Read-cache.
Write-buffer info	Offset and size of the Write-buffer.
Segment size	Fixed size of segments in the OPT.

Table 2: **Borg meta-data.**

Table 2 depicts the OPT meta-data fragment. It stores key persistent information that aid in the operation of BORG. The `BORG_REQUIRE` bit is *set* when the OPT contains data that requires to be copied back to the FS. If set, the operating system initiates BORG at boot time to ensure consistent data accesses. The remaining meta-data information is used to correctly populate the in-memory indirection map structure during BORG initialization.

4 Detailed Design

In this section, we present the design details of BORG by elaborating on its individual components.

4.1 I/O Profiler

The *I/O profiler* is a data collection component that is responsible for comprehensively capturing all disk I/O activity. The I/O profiler generates an *I/O trace* that includes the temporal(timestamp of the request), process(process ID and executable) and the block-level(address range and read/write mode) attributes. We use the **Q** events reported by blktrace [2], which capture the I/O requests queued at the block layer. These include all requests as issued by the file system(s), including any journaling and/or page destaging mechanisms. We defer further details to the blktrace work [2].

4.2 Analyzer

The *analyzer* is responsible for summarizing the disk I/O workload. It first splits the I/O trace obtained from the profiler into multiple I/O traces, one per process. Each process trace is used to build a directed *process access graph* $G_i(V_i, E_i)$, where vertices represent LBA ranges and edges a temporal dependency (correlation) between two LBA ranges. The weight on an edge between vertices (u, v) represents the frequency of accesses (reads or writes) from u to v. The *directed* and *weighted* graph representation is powerful enough to identify repeated sequences of multiple non-sequential requests.

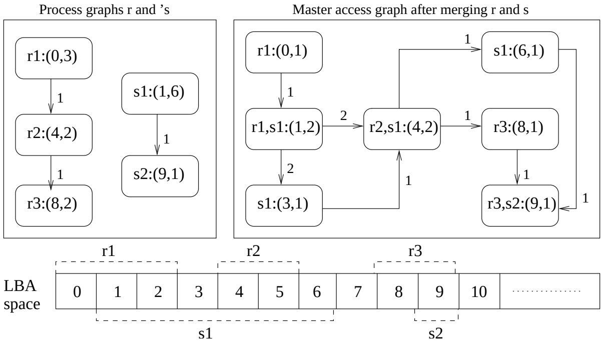


Figure 4: **Building the master access graph.** *Vertices are defined by (start LBA, size of request). Since vertices r_1 and s_1 have overlapping LBAs, r_1 is split into two vertices in the master access graph, one with size 1 and the other with the overlapping s_1 blocks, starting at LBA 1 with size 2.*

Since multiple processes may access the same LBA, a single *master access graph* $G(V, E)$, that captures all available correlations into a single input for the reconfiguration planner is created (illustrated in Figure 4). The complexity of the merge process increases if two vertices (either within the same graph or across graphs) have overlapping ranges. This is resolved by creating multiple vertices so that each LBA is represented in at most one range vertex. While we omit the detailed algorithm for vertex splitting and graph merging due to space constraints, we point out that we reduce the complexity of the merge algorithm by keeping the vertices sorted by their initial LBA. The total time complexity for the analyzer stage is given by $\O(n * l)$, where n is the number of vertices and l is the size of the largest vertex in the graph. Once the merge operation is completed, the master access graph, G, is obtained.

4.3 Planner

The *planner* takes the master access graph, G, as input and determines a reconfiguration plan for the OPT partition. It uses a greedy heuristic that starts by choosing for placement the most connected vertex, u, i.e., with the maximum sum of incoming and outgoing edges (Figure 5). Next it chooses the vertex v most connected (in one direction only, either incoming or outgoing) to u. If v lies on the outgoing edge of u, it is placed after u and if it lies on the incoming edge it is placed before. The next vertex to be placed is the one most connected to the *group* $u \cup v$. This process is repeated until either all the vertices in G are placed, or the OPT is fully occupied, or the edges connecting to the unplaced vertices in the master graph have weight below a certain threshold. If the graph contains disconnected components, each of these are placed as separate groups. The time complexity for the planner is $O(n * lg(m))$ where n is the number of vertices and m is the number of edges; finding the most connected vertex takes $n * lg(m)$ time and finding the next vertex takes $O(m)$ time .

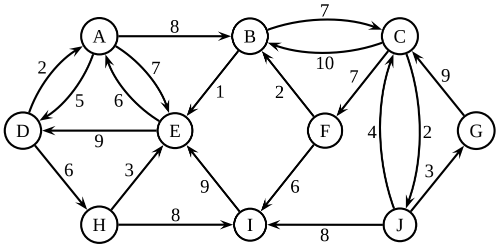


Figure 5: **Placing the master access graph.** *C is the most connected vertex and is chosen for placement first. Next, vertex B is placed $after$ vertex C since it is connected by an outgoing edge and has the highest weight of all the edges connected to C. Next, vertex G is placed $before$ vertex group $C \cup B$. The final sequence of vertices from the lowest LBA to the highest is: $L = [F, H, J, A, G, C, B, E, D]$.*

4.4 OPT Reconfigurator

The *OPT reconfigurator* implements the plan created by the planner component by performing the actual data movement to realize the new configuration of the OPT. This task is complicated primarily because of consistency and overhead concerns. Overhead is partially addressed by issuing low-priority I/O requests[3] for data layout reconfiguration. BORG ensures block data consistency between the FS and OPT copies of data blocks by maintaining a persistent indirection map, termed the `borg_map`, that continuously tracks the most up-to-date location of a data block. This map is updated each time a block location changes.

The reconfigurator copies blocks in three stages; *outgoing*, where it copies all the dirty blocks that are no longer in the new plan back to the original file system (FS) location, *relocate* where, it copies blocks that have to be relocated within the OPT, and *incoming* where it copies all the new blocks that have to be copied from the FS to the OPT. A single data movement operation and the corresponding update on `borg_map` entry can be considered 'atomic' since any application *write* request to the *source* LBA during data movement is put on hold until after the movement is complete and the `borg_map` entry is updated. This ensures that an up-to-date version of data is always maintained by the file system.

4.5 I/O Indirector

The *I/O indirector* operates continuously, redirecting file system I/O requests as required. An I/O request may be composed of arbitrary number of pages. Each page request is handled separately based on (i) number of blocks that can be satisfied from the OPT as per the `borg_map` entry, (ii) type of operation (read or write) and (iii) presence of a free page in the OPT.

For each I/O request larger than one page, the indirector splits it into multiple per-page requests. If mapping exists for all the pages of the I/O request in the `borg_map`, the request is indirected to the OPT. If no mapping exists, and the request is a read request, it is issued unchanged to the file system. If only some pages of a read I/O request are mapped and the mapped entries are clean, the entire I/O is indirected to the file system; this optimization reduces the seek overhead incurred to serve the request partially from the OPT and the rest from the FS. For a write request, when no mapping exists for any of the pages, the blocks are still written to a *write-buffer* portion of the OPT reserved for assimilating write requests (if space permits) and a new entry is created in the `borg_map` with the dirty bit set. For a partially-mapped write request, the mapped blocks are indirected to their OPT locations; the unmapped pages are also absorbed in the write-buffer, space permitting, otherwise these are issued to the FS.

4.6 Kernel Data Structures

The persistent data structure `borg_map` is implemented as a radix tree such that given an FS LBA, the OPT LBA can be retrieved efficiently and vice-versa. It also maintains the *dirty* information for the OPT LBAs. For every page of 4KB, BORG stores 4 bytes each for the forward and the reverse mapping and one dirty bit. If all the pages in the OPT of size S GB are occupied, the worst case memory requirement is $2 * S$ MB (S MB for forward and reverse mapping each), and $\frac{S}{2^5}$ MB for the dirty information. Thus, in the worst case, `borg_map` requires memory of 0.25% of the size of the OPT partition, a typically acceptable requirement for kernel-space memory.

5 Implementation Issues

In this section, we discuss the particularly challenging aspects of the BORG implementation that help address data consistency and overhead.

5.1 Persistent Indirection Map

Since BORG replicates popular data in the OPT space, the system must ensure that reads are always up-to-date versions of data, including after a clean shutdown or a system crash. BORG implements a persistent `borg_map`, which is distributed within read-cache and

---

[3] Priority scheduler is a prerequisite. BORG prototype uses CFQ.

write-buffer segments of the OPT. Map entries on-disk are updated (along with their in-memory version) each time the OPT partition is reconfigured or when a new map entry is added to accommodate a new write absorbed into the OPT. Upon writes to an existing OPT mapped block, its indirection entry in the in-memory copy of the reconfiguration map is marked as dirty, once the I/O is completed. To minimize overhead for OPT writes, we chose not to maintain dirty information in the on-disk copy. Upon reboot after an unclean shut down, all entries in the persistent map are marked as dirty and future IOs to these blocks are directed to the OPT.

5.2 Indirection Complexity

BORG maintains metadata at the granularity of a *page* (rather than *block*) to reduce metadata memory requirement (by 8X for Linux file systems). Consequently, the indirector must carefully handle I/O requests whose sizes are not multiples of the page-size and/or which are not page-aligned to the beginning of the target partition. Figure 6 illustrates this problem. We address this issue via I/O request splitting and page-wise indirection.

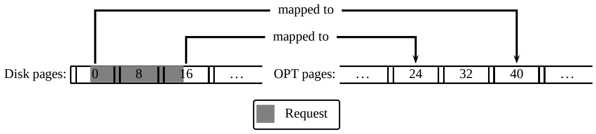


Figure 6: **Addressing request alignment during indirection.** *Two pages of the FS are mapped to the OPT. The first FS page starting at block 0 is mapped to the fifth OPT page starting at block 40. Similarly, the third FS page is mapped to the fourth OPT page. The second FS page is not mapped to the OPT. The application I/O request is the shaded region.*

For splitting I/Os, BORG creates new requests using the Linux kernel *block io* structure called `bio`, one per page. All attributes of the `bio` structure are then populated based on the *indirection map*, including the sector, offset, and length within the page that will be filled/emptied depending of the operation. After the splitting and issuing each “sub-I/O”, the indirector waits for all sub-I/Os to complete before notifying the application. A special case arises while handling write requests that are not already mapped to OPT and are not page-aligned. To avoid partially inconsistent OPT pages, we let such requests continue on to the file system partition.

5.3 Optimizing Reconfiguration

Consider a set L of n LBAs, $L_1, \cdots, L_n$, sequentially located in the OPT space. L forms a *chain* if $\forall L_i \in L$, where $L_i \neq L_n$, L_i has to be relocated to location $L_i + 1$ and L_n an outgoing block. If L_n, has to be relocated to L_1 within the OPT, L forms a *cycle*. Information about chains and cycles, that occur exclusively for the relocated blocks, can be used to further optimize data movement. If a cycle exists, it is broken by copying the last block L_n back to the FS (if dirty) and then deleting the plan entry for that block; an additional plan entry is then created to mark this as `incoming` block to L_o. Next, all remaining blocks belonging to the same chain/cycle are copied to their new locations in the OPT. To do so, the reconfigurator issues all reads to the source locations in parallel; once all reads have been completed, it issues all the writes in parallel, in each case allowing the I/O scheduler to optimize the request schedule.

5.4 Module Unload

BORG is dynamically included in the I/O stack by substituting the `make_request` function of the device targeted for performance optimization. While module insertion is straightforward, module removal/unload must ensure data consistency. Upon removal, BORG flushes dirty OPT blocks to their original locations in the file system. In addition, BORG must address race conditions caused when an application issues an I/O request to a page that is being flushed to disk at that exact instant. BORG stalls (via `sleep`) the foreground I/O operation until the specific page(s) being flushed are written to the disk. Since we expect module unload to be a rare event and the probability that a request for a page at the exact time it is being flushed to be low, the average response time for application I/O remains virtually un-impacted.

6 Evaluation

We evaluate BORG under various workloads to answer the following questions.

(i) How well does BORG perform? We use disk bandwidth observed during active I/O operation (i.e., excluding idle periods) as our primary performance metric to evaluate the performance benefits of BORG with different kinds of workloads. We compare performance against a `vanilla` system when all the blocks are statically located in the FS space.

(ii) Why is BORG effective? We would like to know if BORG gains mainly because of the sequentiality or the proximity of data (or both) in the OPT. We use two metrics, *non-sequential accesses percentage*[4] and *average seek distance* for a workload for this purpose.

(iii) When is BORG not effective? BORG can degrade the system performance for certain workloads. We evaluate BORG for varying workloads and when it would start performing worse than the vanilla system.

(iv) How much CPU resource overhead does BORG incur? While the upper bound on memory overhead was elicited in § 4.6, the CPU resources consumed by BORG

---

[4] Measured as $\{\frac{numberOfSeeks}{totalBlocksRead}\}$. Non-sequential accesses are an order of magnitude (or more) less efficient than sequential; even a small reduction in this metric can lead to substantial performance benefit.

Host	Make	Model	RAM (MB)	Capacity (GB)		
				Total	FS	OPT
`O1`	WD	2500AAKS	1024	250	46	1
`O2`	WD	360GD	1024	39	24	2
`O3`	Maxtor	6L020L1	1024	20	15	2
`O4`	WD	2500AAKS	1024	250	180	8
`O5`	Maxtor	6L020J1	1536	20	8	1

Table 3: **Experimental test-bed details.**

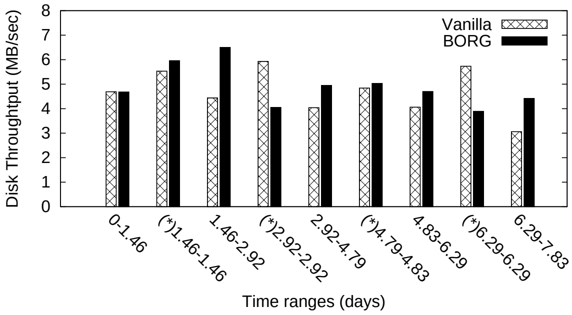


Figure 7: **Disk throughput in various phases of the developer trace replay.** *(*) Ranges that correspond to reconfiguration phases.*

should also be within acceptable limits. We use the execution times for various stages of BORG as an approximate measure of CPU resource utilization.

Experimental Setup. All experiments were performed on machines running the Linux 2.6.22 kernels. We used host machines, `O1` through `O5`, with differing hardware configurations and disk drives (Table 3). We used `reiserfs` for `O1` and `O3`, and `ext3` for the rest. No additional hardware was required to implement BORG.

We conducted four different set of experiments. The first set use week-long traces of a developer's system and an SVN server. The second experiment is an actual deployment of a web server that mirrors our CS department's web server. The third experiment evaluates BORG performance in a virtual machine environment. The fourth experiment evaluates the performance improvement due to BORG for application start-up events.

6.1 Trace Replay

To evaluate BORG under realistic workloads, we conducted trace replay experiments using *SVN server* and *developer* workloads described in Table 1. For the traces and the replay, we used *blktrace* and *btreplay* respectively [2]. Additionally for the developer workload, we evaluate the impact of the write buffer size in BORG. We used an acceleration factor of 168x, after verifying that the resultant block access sequence was not affected due to the speed-up. In each experiment, we performed 4 reconfigurations, equally spaced in time; the trace-playback acceleration factor was reverted to 1x during each reconfiguration operation to accurately estimate the impact of reconfiguration overhead on foreground I/O.

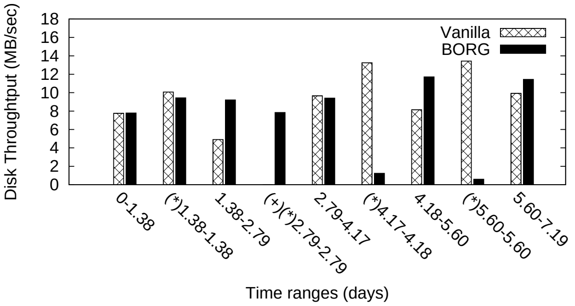


Figure 8: **Disk throughput in various phases of the SVN server trace replay.** *(*) Ranges correspond to reconfiguration phases. (+) The vanilla bar here is zero because there are no I/Os for that range.*

6.1.1 SVN Server

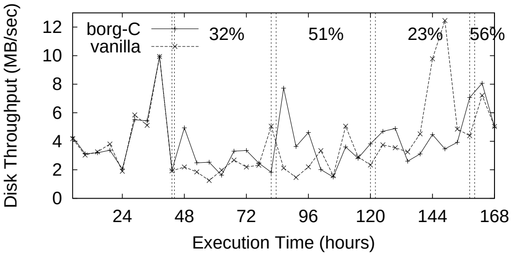
For the SVN server trace replay, we used the host *O2* (Table 3). The write buffer size was set to 20% of the OPT size. Figure 8 shows the disk throughput of the replay in different phases of the experiment. In all the reconfiguration phases the throughput, as expected, is notably lower. However, it is clear that in almost all the non-reconfiguration phases, BORG behaves better than the vanilla configuration. In the best case (range 1.38-2.79), BORG improves the disk throughput by approximately 50%. This is a surprising result, since as per Figure 1(c), the working-set for this workload undergoes rapid shifts. One explanation is that a write-intensive workload and the OPT write-buffer is successful in sequentializing a rapidly changing, possibly non-sequential, write workload. Analysis of the block level traces revealed that with borg, the non-sequential access percentage reduced from 1.70% to 1.15%, and the average seek distance reduced from 704 to 201 cylinders.

6.1.2 Developer

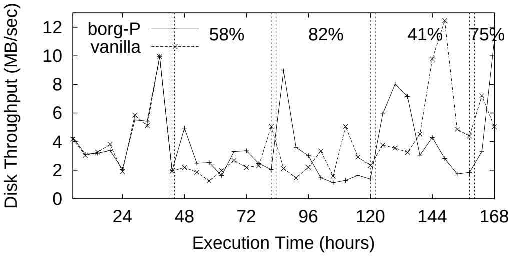
For the developer trace replay, we used the host *O1* (Table 3) with the OPT write buffer set to 40% of the OPT size. Figure 7 shows the disk throughput for this experiment in all the BORG phases. For the reconfiguration phases, once again, the throughput of the BORG configuration is substantially lower. In contrast to the SVN server workload, we see a clear increase of performance (13% to 30%) for all the non-reconfiguration periods of time. Analysis of the block level traces revealed that with BORG the non-sequential access percentage reduced from 3.93% to 3.30%, and the average seek distance reduced from 1203 to 782 cylinders.

6.2 Web Server

To evaluate BORG in a production server environment, we made a copy of the our Computer Science department web server on the *O4* machine (see Table 3), and replayed all the web requests for a one week period. During this week a total of 1137234 requests to 256017 distinct files were serviced. We set BORG to reconfigure four times during this period, using an OPT of 8GB ($<$ 5% of the 170GB web server file system). To measure the influence of the I/O history, we conducted two sets of experiments. In the first experiment, we used all the traces gathered from the beginning of the experiment as input to the reconfigurator (cumulative). For the second, we only used the portion of the trace corresponding to period since the last reconfiguration (partial).



(a) Using cumulative trace



(b) Using last interval's trace

Figure 9: **Disk throughput for the week long web log replay.** *Dotted vertical lines represent the start and ending of the I/O activity of a reconfiguration, and the percentages are the hit ratio to the OPT in that interval.*

Figure 9 shows the improvements in disk throughput across various phases of the experiment along the 1 week timeline when compared to a vanilla system. For both the *cumulative* and the *partial* experiments, BORG improves disk throughput in most cases, but not all, particularly performing worse approximately two-thirds into the week-long trace. This is partially explained by the lower OPT hit-ratios approximately two-thirds into the experiment, suggesting a more sharply changing working-set. When considering the overall average, BORG outperforms the vanilla system, yielding improved average throughput of 12% and 17%, for the cumulative and partial trace experiments respectively (not shown in figure). It is interesting to note that short term training yielded better results in this case, perhaps due to greater influence of short term locality.

Next we examine operational overhead of BORG. Figure 10 shows the amount of time taken in each phase of the reconfiguration. With cumulative traces, the time required for the analyzer and planner phases increases linearly. While the planner and analyzer stages can run as low-priority tasks in the background, we must point out that the current implementation of BORG analyzer and planner stages are highly unoptimized and there is substantial room for improvement. Specifically, both the planner and analyzer fail to take advantage of their past computations altogether, which is the primary reason for the linear increase in their processing times. We believe that these overheads can be reduced in the future to exhibit sub-linear (and possibly constant) behavior. With partial traces, the time increases until the second reconfiguration, but then decreases and stays almost constant for the following ones.

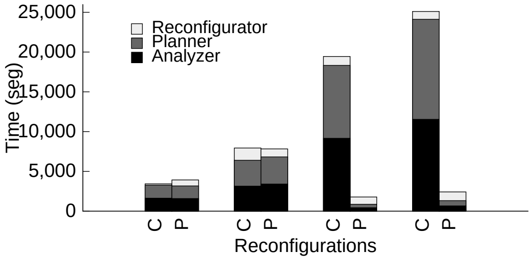


Figure 10: **BORG overhead.** *Column* C *and* P *represent the cumulative and partial traces experiments respectively.*

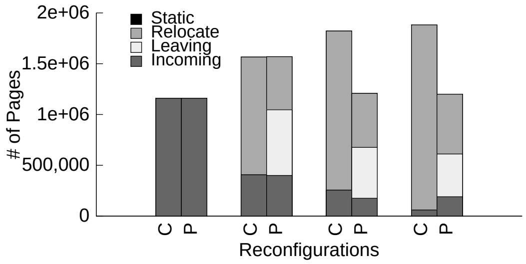


Figure 11: **Differences in the reconfiguration plans.**

To explain this phenomena we examine the reconfiguration plan divided by the type of operation (refer to § 4.3), presented in Figure 11. We note that the size of the plan constantly increases for the cumulative case and most of the movements correspond to page relocates, that is changes within the OPT. The story is very different for partial traces, where we see unused pages leaving the OPT, maintaining a smaller working set in the OPT and thereby reducing the amount of work done by subsequent analyzer and planner stages.

6.3 Virtual Machines

BORG has the potential to significantly improve the performance of virtualized environments, by capturing multiple virtual machine (VM) localities spread across a physical volume. We evaluated the impact on the per-

VM boot time and the overall performance of virtual machines by deploying BORG in a Xen [3] virtual machine monitor. We created four VMs, each with 64MB memory and 4GB physical partition on the host *O5* (refer to Table 3). For evaluating boot-time improvement, we trained BORG with the boot-time events of all the virtual machines. BORG showed an almost 3X average improvement in VM boot-times - *167 seconds* with Vanilla and *65 seconds* with BORG.

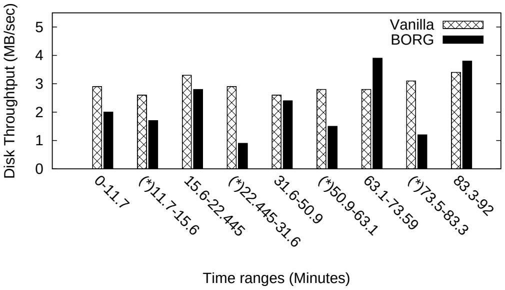


Figure 12: **BORG inside a VMM.** *(*) Ranges correspond to reconfiguration phases.*

To measure normal execution performance improvement for the VMs, we ran the Postmark benchmark which emulates an e-mail server and creates and updates small files. We set the number of files to be 2000 in 500 directories and performed 200,000 transactions. We reconfigured BORG after every 20% of the benchmark was executed with the training set including I/O operations from the start of the execution of the benchmark. The results for the I/O performance are shown in Figure 12. As before, the reconfiguration phases see a reduced disk throughput with BORG. For the normal operation, as the training set increases, the throughput with BORG starts improving. The overall performance improvement is modest with the average throughput over the entire execution increasing from 3.0 MB/s to 3.4 MB/s (or 13.3%). However, this improvement is not consistent; performance degrades substantially even during normal operation in the early stages of the benchmark. The *loss of process context* inside the VMM is a key problem that tends to convert sequentially laid out files into non-sequential upon reconfiguration. We believe that making BORG aware of process context inside the VMM [13] can substantially improve the OPT layout.

6.4 Application Start-up

We evaluated the impact of BORG on I/O-bound start-up phase for common desktop applications using host *O3*. We first trained the system for a duration of approximately four hours, during which we invoked a subset of the applications listed in Table 4 (but specifically excluding `gimp` and `ooimpress`) multiple times for performing common office tasks. We invalidated the page cache periodically to artificially dilate time and simulate system reboots. Table 4 shows the difference in application start-up times, the percentage of sequential accesses and average seek overhead. For the applications which were used in training, it can be observed that there is a noticeable improvement in the IO time with BORG - at least 43% for `oowriter` and up to 67% for `eclipse`. Further, it is interesting to observe that although the percentage of sequential I/Os decreases for `oowriter` and `acroread` with BORG, there is an overall improvement in I/O performance, possibly due to a reduction in the rotational overhead . There is barely any difference in the performance for untrained application `gimp`. However, although `ooimpress` was not used in the training, its start-up user-time shows an improvement of 62% in the average IO time; this can possibly be attributed to large shared libraries in common with `oowriter` and which was used in the training.

App	**Start-up time**		**Rand. I/O %**		**Avg seek (#cyl)**	
	V	B	V	B	V	B
`firefox`	3.71	2.32	2.7	1.2	132	37
`oowriter`	5.30	2.74	3.8	0.2	193	20
`xemacs`	7.26	2.72	2.1	0.3	87	9
`acroread`	6.20	2.65	4.6	0.1	39	9
`eclipse`	4.12	1.52	2.5	0.3	198	29
`gimp`	3.62	3.66	2.5	2.1	102	63
`ooimpress`	5.18	1.97	2.7	0.3	61	39

Table 4: **Application start-up time improvement.** *V: vanilla, B: BORG.*

6.5 Sensitivity Analysis

To gain maximum performance improvement with BORG its configurable parameters – the reconfiguration interval, the OPT size, and the OPT write buffer fraction — must be carefully configured for a given workload. We now perform a sensitivity analysis using two of the workloads to better understand the effects of these parameters.

We replayed the developer and the SVN workload traces on host `O1` varying each of these parameters over a range of values. In all the experiments, the trace replay begins at the same starting point, that is after a *base reconfiguration*, which uses the first six hours of the trace as the training period.

6.5.1 Reconfiguration Interval

Figure 13 shows the percentage improvement over the vanilla system. The reconfiguration interval is varied from 8 hours (18 reconfigurations) to 3 days (1 reconfiguration). To boostrap the sensitivity analsyis, the OPT size is fixed to 1GB, with 50% reserved for write buffering in this experiment. The throughput values shown are for the periods inter-reconfiguration intervals, to more accurately estimate improvements. For the developer workload, as the reconfiguration interval increases the throughput increases, the training set becomes larger, and

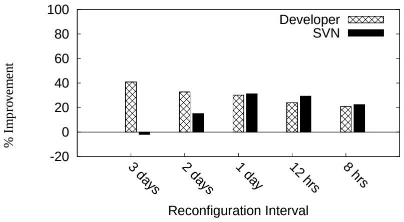


Figure 13: Throughput improvement varying reconfiguration intervals.

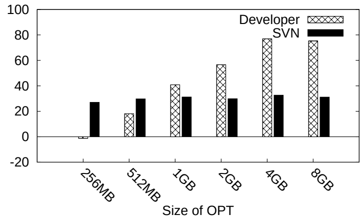


Figure 14: Throughput improvement varying OPT size.

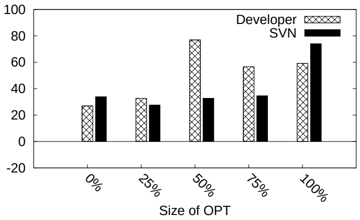


Figure 15: Throughput improvement varying OPT write buffer.

BORG can more effectively capture the working-set. For the SVN workload, the performance decreases for higher intervals. This is because the SVN working-set changes quite frequently (elaboration in § 2 and Figure 1(c)).

6.5.2 OPT size

We use the best-case reconfiguration intervals of 3 days for the developer and a day for the SVN workload from the previous experiment. We vary the OPT size from 256MB to 8GB, of which the write buffer is always chosen as 50% of the OPT size. Figure 14 shows that as the OPT size increases, BORG's performance with the developer workload increases since the developer workload has a larger working set. When most of the blocks in the working set can be accomodated in the OPT, the performance improvement stabilizes. Since the working set size for the SVN workload is relatively smaller, the performance improvement is almost same for the OPT sizes $>$256MB.

6.5.3 Write Buffer Variation

From our previous results, we pick an interval of 3 days and 1 day and OPT size of 2BG and 4GB for the developer and the SVN workloads respectively. We vary the write buffer from 0-100%. Figure 15 shows that for the developer workload, not having a write buffer results in the lowest throughput. There is a steady increase in performance, peaking at 50% write buffer. Thereafter, it starts falling since read performance begins to degrade due to lesser available read cache. For the write-intensive SVN workload, the performance increases with increase in the write buffer size, since all the writes can be colocated in the OPT partition.

Configuring BORG parameters The above experiments indicate that configuring parameters incorrectly can lead to sub-optimal performance improvements with BORG. Fortunately, iterative algorithms can be easily employed to identify better parameter combinations in a straightforward way. Exploring such iterative algorithms more formally is one aspect of our future work.

7 Related Work

We examine related work by organizing the literature into block and file based approaches.

7.1 Block level approaches

Early work [38] on optimized data layout argued for placing the frequently accessed data in the center of the disk. Vongsathorn *et al.* [36] and Ruemmler and Wilkes [27] both propose *Cylinder Shuffling*. Ruemmler and Wilkes specifically demonstrated that performing relatively infrequent shuffling led to greater improvement in I/O performance. In Akyurek and Salem's work [1], the authors demonstrated the advantages of *copying* over *shuffling* and the importance of reorganization at the block (rather than cylinder) level. These early *data clustering* approaches emphasized on process- and access-pattern- agnostic *block counts* to perform the data reorganization and reported simulation-based results.

Researchers have also investigate self-optimizing RAID systems. Wilkes *et al.* proposed HP AutoRAID [37], a controller-based solution, that transparently adapts to workload changes by using a two-level storage hierarchy; the upper level provides data redundancy for popular data while the lower level provides RAID 5 parity protection for inactive data. Work on eager writing [39] and distorted mirrors [33] address mirrored/striped RAID configurations primarily for database OLTP workload (which are characterized by little locality or sequentiality) that choose to write to a free sector closest to the head position on one more disk drives. While we are yet to explore BORG's use in multi-disk systems, the optimizations used in BORG are different and mostly complementary to the above proposals, whereby BORG attempts to capture longer-term on-disk working-sets within a dedicated volume.

Hu *et al.*'s work on Disk Caching Disk [9] uses an additional logging disk (or disk partition) to perform writes sequentially and subsequently, destage to their original locations. Write buffering in BORG is slightly different in that writes to data already in the OPT partition are written in place. The DCD work does not optimize for data read operations; BORG optimizes reads as well so head movement is substantially restricted.

Among recent work on block reorganization, Salmon *et al.* [29] describe a framework for combining multiple heuristics for data reorganization. BORG could easily fit within this larger framework. C-Miner [16] uses advanced data mining techniques to mine correlations between block I/O requests. These techniques can be utilized in BORG to infer complex disk access patterns. The Intel Application Launch Accelerator [11] reorganizes blocks used during application start-up to be more sequential, but does not provide a generic solution to improve overall disk I/O performance of the system.

Among block level approaches, our work is closest to ALIS [8], wherein frequently accessed blocks as well as block sequences are placed sequentially on a dedicated, *reorganized area* on the disk. There are key differences in design and implementation, though. First, BORG incurs reduced space, maintenance, and metadata overhead since it maintains at most one copy of each data block. The multiple replicas in ALIS can become stale quickly in write-intensive workloads. Further, unlike BORG, ALIS does not optimize write traffic. Finally, the evaluation of ALIS techniques is performed using a disk simulator with trace playback. On the other hand, we implement and evaluate an actual system, thereby having the opportunity to address a greater detail of system implementation issues.

7.2 File level approaches

In one of the early file oriented approaches, Staelin *et al.* [34] proposed monitoring file accesses and moving frequently accessed files (entirely) to the center of the disk. Log-structured file systems (LFS [26]) offer superior performance for workloads with large number of small writes by batching disk writes to the end of a disk-sequential *log*. BORG writes all data to the OPT partition to achieve a similar effect, but also attempts to co-locate a majority of read operations with the writes. Matthews *et al.* [17] proposed an optimization to LFS by incorporating data layout reorganization to improve read performance. Their use of *block access graphs* is similar to the *process access graphs* used in BORG. Their LFS-specific solution moves blocks within the LFS partition storing exactly one copy of each block at any time. Since BORG stores two copies, it can optimize for sequential and application-driven deterministic, non-sequential accesses simultaneously.

Researchers have also explored data- and application-specific layout mechanisms. Ganger and Kaashoek [5] advocate collocating inodes and file blocks for small files. Conversely, PLACE [21], exposes the underlying layout structure to applications, so they can perform custom data placement. Sivathanu *et al.* [32] propose semantically-smart disk systems (SDS) that infer file system semantic associations for blocks, subsequently used for aligning files with track boundaries. Windows XP [19] uses the defragmenter for co-locating temporally correlated file data for speeding up application start-up events. BORG is a generic solution in comparison to the above approaches, since creates a block reorganization mechanism that can adapt to an arbitrary workload.

Among file level approaches, BORG is closest to the FS2 [10]. FS2 proposes replication of frequently accessed blocks based on disk access patterns in file system free space. This strategy, unfortunately, also restricts the degree of seek and rotational-delay optimization due to the distribution of free space. Since FS2 may create multiple copies of a block simultaneously, staleness, and consequently, space and I/O bandwidth wastage, become important concerns (similar to those in ALIS); BORG maintains at most one extra copy of each block and its strength is in being a non-intrusive, storage-stack friendly, and file system independent (portable) solution.

8 Conclusions and Future Work

We presented BORG, a self-optimizing layer in the storage stack that automatically reorganizes disk data layout to adapt to the workload's disk access patterns. BORG was designed to optimize both read and write traffic dynamically by making reads and writes more sequential and restricting majority of head movement within a small optimized disk partition. A Linux implementation of BORG was evaluated and shown to offer performance gains in the average case for varied workloads including office and developer class end-user systems, a web server, an SVN server, and a virtual machine monitor. Average disk throughput improvement with BORG across these workloads range from 13.3% (for the VM workload) to 50% (for the SVN server workload).

BORG performs occasionally worse than a vanilla system, specifically when a read-mostly workload (e.g., a web server) drastically shifts its working set. BORG is able to easily address changing working-sets with a (possibly non-sequential) write workload (e.g. an SVN server), since it has the ability to absorb and sequentialize writes inside the OPT. A sensitivity analysis revealed the importance of choosing the right configuration parameters for reconfiguration interval, OPT size, and the write-buffer fraction. Fortunately, simple iterative algorithms can be quite effective in identifying the right parameter combination; a formal investigation of such an approach is an avenue for future work. The memory and CPU overheads incurred by BORG are modest, and with ample scope for further optimization. In summary, we believe that BORG offers a novel and practical approach to building self-optimizing storage systems that can offer large I/O performance improvements in commodity environments.

References

[1] S. Akyurek and K. Salem. Adaptive Block Rearrangement. *Computer Systems*, 13(2):89–121, 1995.

[2] J. Axboe. blktrace user guide, February 2007.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and

the art of virtualization. In *Proc. of the ACM SOSP*, pages 164–177, New York, NY, USA, 2003. ACM.

[4] H. Custer. Inside the Windows NT File System. *Microsoft Press*, August 1994.

[5] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. *Proc. of the USENIX Technical Conference*, 1997.

[6] M. Gómez and V. Santonja. Characterizing Temporal Locality in I/O Workload. *Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2002.

[7] M. Holton and R. Das. XFS: A Next Generation Journalled 64-bit filesystem with Guaranteed Rate IO. *SGI Technical Report*, 1996.

[8] W. W. Hsu, A. J. Smith, and H. C. Young. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems*, 23(4):424–473, Nov 2005.

[9] Y. Hu and Q. Yang. DCD – Disk Caching Disk: A New Approach for Boosting I/O Performance. *Proc. of the International Symposium on Computer Architecture*, 1995.

[10] H. Huang, W. Hung, and K. G. Shin. FS2: Dynamic Data Replication In Free Disk Space For Improving Disk Performance And Energy Consumption. *Proc. of the ACM SOSP*, October 2005.

[11] Intel Corporation. Intel application launch accelerator. *Online at http://support.intel.com/support/chipsets/iaa/*, 1998.

[12] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. *Proc. of the ACM SOSP*, September 2001.

[13] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. *Proc. of the USENIX Technical Conference*, May 2006.

[14] D. Kleikam, D. Blaschke, S. Best, and B. Arndt. JFS for Linux. *http://jfs.sourceforge.net/*.

[15] C. Li and K. Shen. Managing Prefetch Memory for Data-Intensive Online Servers. *Proc. of the 4th USENIX FAST*, pages 253–266, December 2005.

[16] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou. C-Miner: Mining Block Correlations in Storage Systems. *Proc. of the USENIX FAST*, pages 173–186, April 2004.

[17] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. *Proc. of the ACM SOSP*, 1997.

[18] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX*. *ACM Transactions on Computer Systems 2*, 3:181–197, August 1984.

[19] Microsoft Corporation. Fast System Startup for PCs Running Windows XP. *Windows Platform Design Notes*, December 2006.

[20] Namesys, Inc. The ReiserFS File System. *Go to: http://www.namesys.com/*.

[21] J. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. *Proc. of the USENIX Annual Technical Conference*, pages 311–324, June 2003.

[22] A. E. Papathanasiou and M. L. Scott. Aggressive Prefetching: An Idea Whose Time Has Come. *Proc. of the Workshop on HotOS*, June 2005.

[23] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th ACM SOSP*, December 1995.

[24] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. *Proc. of the USENIX Annual Technical Conference*, pages 297–310, June 2003.

[25] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). *Proc. of the IEEE RTSS*, December 2003.

[26] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *Proc. of the ACM SOSP*, October 1991.

[27] C. Ruemmler and J. Wilkes. Disk Shuffling. *Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories*, October 1991.

[28] C. Ruemmler and J. Wilkes. UNIX disk access patterns. *Proc. of the Winter USENIX Conference*, 1993.

[29] B. Salmon, E. Thereska, C. Soules, and G. Ganger. A Two-tiered Software Architecture for Automated Tuning of Disk Layouts. *Workshop on Algorithms and Architectures for Self-Managing Systems*, 2003.

[30] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. *Proc. of the Winter 1990 USENIX Technical Conference*, 1990.

[31] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Operating Systems. *Proc. of the Workshop on HotOS*, May 1997.

[32] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. *Proc. of the USENIX FAST*, March 2003.

[33] J. A. Solworth and C. U. Orji. Distorted Mirrors. pages 10–17, 1991.

[34] C. Staelin and H. Garcia-Molina. Smart Filesystems. In *USENIX Winter Conference*, pages 45–52, 1991.

[35] S. C. Tweedie. Journaling the Linux ext2fs File System. *The Fourth Annual Linux Expo*, May 1998.

[36] P. Vongsathorn and S. D. Carson. A System for Adaptive Disk Rearrangement. *Softw. Pract. Exper.*, 20(3):225–242, 1990.

[37] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *Proc. of the ACM SOSP*, 1995.

[38] C. K. Wong. Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems. *ACM Computing Surveys*, 12(2):167–178, 1980.

[39] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. *Proc. of the USENIX File and Storage Technologies*, January 2002.